Knapsack problem: Implementation of an Optimization Algorithm based on the Operational Research techniques for Complexity Analysis.

Hugo Silva, Jaques Resende, Luís Oliveira

[a] Instituto Politécnico do Porto, Escola Superior de Tecnologia e Gestão, Rua do Curral, Casa do Curral– Margaride, 4610-156 Felgueiras, Portugal.
8180378@estg.ipp.pt, 8190214@estg.ipp.pt, 8190370@estg.ipp.pt

Abstract. An example of a classic problem in the field of operational research and combinatorial optimization is the knapsack problem. This report describes the knapsack problem then presented the dynamic programming algorithm as a solution and analyze it complexity.

Keywords: knapsack, problem, dynamic, programming, algorithm, complexity

1. Introduction

The knapsack problem is a NP (nondeterministic polynomial) problem optimization that has been studied since 1897[1] and named by the mathematician Tobias Dantzig[2].

The quantity of each item to include in the knapsack is determined by the given weight of *n* items. The objective is so that the value is as large as possible, and the weight isn't higher than the given limit.

This problem can be applied in resource allocation, project selection, cargo packing and other fields.

In this report we will explain how we chose and implemented an optimization algorithm base on the operational research techniques for complexity analysis to solve the knapsack problem.

2. Materials and methods

We chose to code the algorithm in Java[4] programming language using the Gradle plugin to create tests and validate the algorithm complexity.

To efficiently solve the Knapsack problem, the group searched the web and arrived at the conclusion that the best method was a dynamic programming algorithm [3]. This is an algorithmic technique that breaks the problem into simpler and smaller subproblems until a subproblem that can be solved easily is found.

In this case, each subproblem solution is stored in a table until the main problem is solved (Attachment: Output Table).

Code Explanation:

$W_{[i]}$ is the item weight and $V_{[i]}$ is its value, each item is optional and can be put in the Knapsack and only integers quantities are accepted.

This problem depends on two conditions: the number of packages and the remaining weight that can be stored. With dynamic programming, we have the objective function that depends on two variables and a two-dimensional table of solutions (variable: *B)*

When there are *i* items and the weight limit *M*, $B_{[i][j]}$ is the optimal weight while the maximum weight of the knapsack is *j*. The optimal weight is less than or equal to maximum weight - $B_{[i][j]} \leq j$.

When there is only one item to pick, $B_{[i][j]}$ is calculated for each *j*.

$B_{[1][j]} = W_{[1]}$

Since *j* is the weight limit, there can be found two possible optimal solutions among the items.

1- The item *i* isn't picked, the maximum possible value is $B_{[i][j]}$.

$$B_{[1][j]} = B_{[i-1][j]}$$

2- The selected item is *i so* $B_{[1][j]}$ *is* equal to the value $V_{[i]}$ of item *i* plus the maximum value can be found by choosing amidst item with weight limit.

$$B[i][j] = V[i] + B[i-1][j-W[i]]$$

The max of the above two values will be $B_{[i][j]}$.
To determine if it's better to pick the item or not we use the following recursive formula:

$$B_{[i][j]} = max(B_{[i-1][j]}, V_{[i]}+B_{[i-1][j-w[i]]}$$

Based on this formula a possibilities table with $n+1$ lines and $M+1$ columns is built.
Line 0 is filled with zeros. After that, the line is used to determine line 1 through recursion, line 1 to determine line 2 and the same happens with the next lines.
The goal of the possibilities table is to find the maximum value $B_{[n][j\ M]}$ when selection in all n packages with the weight limit M.
If $B_{[n][M]} = B_{[n-1][M]}$ means that the item n is not picked and $B_{[n-1][M]}$ is selected else if $B_{[n][M]} \neq B_{[n-1][M]}$ then the package is on the optimal selection and $B_{[n-1][M-W[n]]}$ is selected and this continues till row zero of the possibilities table.
To find the selected packages in the possibilities table we used an algorithm that follows the steps bellow:

- $i=n, j=M$.
- find the line I that $B_{[i][j]} > B_{[i-1][j]}$. Flag picked package i: Select [i] = true.
- $j = B_{[i][j]} - W_{[i]}$. If $j > 0$, go to previous step, otherwise go to next step.
- Show the picked items based on the possibilities table.

The code:
(see attachment: Knapsack method and MAIN)
Regarding the code, table *B* is generated, and each cell has 0 as default value.

```
int[][] B = new int[n + 1][M + 1];
```

Build the table B in bottom-up manner (looking first at the smaller subproblem and then, using the solution found, solve the larger ones.

```
for (int i = 1; i <= n; i++)
```

Determine $B_{[i][j]}$. Package *i* is selected on not.

```
B[i][j] = B[i - 1][j];
```

Check if picking item *i* will be more beneficial then reset $B_{[i][j]}$.

```
if ((j >= W[i - 1]) && (B[i][j] < B[i - 1][j - W[i - 1]] + V[i - 1]))
```

Trace the table from row n to row 0.

```
while (n =! 0)
```

If package n is selected then can only add weight $M - W_{[n-1]}$.

```
M = M – W[n – 1]
```

The code also has the variable *comp* that counts the number of comparisons that occur during the runtime of the algorithm. This variable will be used to calculate the algorithm complexity. This way, the analysis is independent of the machine running it.

In each iteration the algorithm verifies the first comparation an then the *comp* variable is incremented. If false, no more comparisons are done, if true the second comparison is verified and *comp* is incremented again.

In each test battery, four sets of input instances were used. The size of the next instance is twice the size of the one before. The sizes used were 4, 8, 16 and 32. The maximum backpack weight used on the tests were 150 and 1000.

To calculate the complexity order ($\boldsymbol{O(f(n))}$) we compute and record the value of the *comp* and find the most similar with the values of standard complexity function *f(n)* [3].

3. Experimental Results

The next tables show the results that were obtained. In the first column we have the input instance size and in the second the test *comp* value obtained. The following columns have the standard complexity orders values expected [5]. (see attachment: Standard Complexity Functions)

Table 1 – *comp* value vs Complexity Order, weight = 150

Input instance size (n)	*z*	$O(1)$	$O(n)$	$O(n^2)$	$O(\log_2 n)$	$O(n\log_2 n)$	$O(n^3)$	$O(n^4)$	$O(2^n)$
4	968	968	968	968	968	968	968	968	968
8	2058	968	1936	3872	969	984	7744	15 488	937024
16	4183	968	3872	15 488	970	1032	61952	247 808	$8{,}78014*10^{11}$
32	8382	968	7744	61952	971	1160	495 616	3 964 928	$7{,}70909*10^{23}$

Table 2 - comp value vs Complexity Order, weight = 1000

Input instance size (n)	*Comp Value*	$O(1)$	$O(n)$	$O(n^2)$	$O(\log_2 n)$	$O(n\log_2 n)$	$O(n^3)$	$O(n^4)$	$O(2^n)$
4	7948	7948	7948	7948	7948	7948	7948	7948	7948
8	15862	7948	15 896	31 792	7949	7964	63 584	127 168	63 170 704
16	30545	7948	31 792	127 168	7950	8012	508 672	2 034 688	$3{,}99054*10^{15}$
32	60224	7948	63 584	508 672	7951	8140	4 069 376	32 555 008	$1{,}59244*10^{31}$

Graphical comparison of tests comp value obtained and values of standard complexity function:

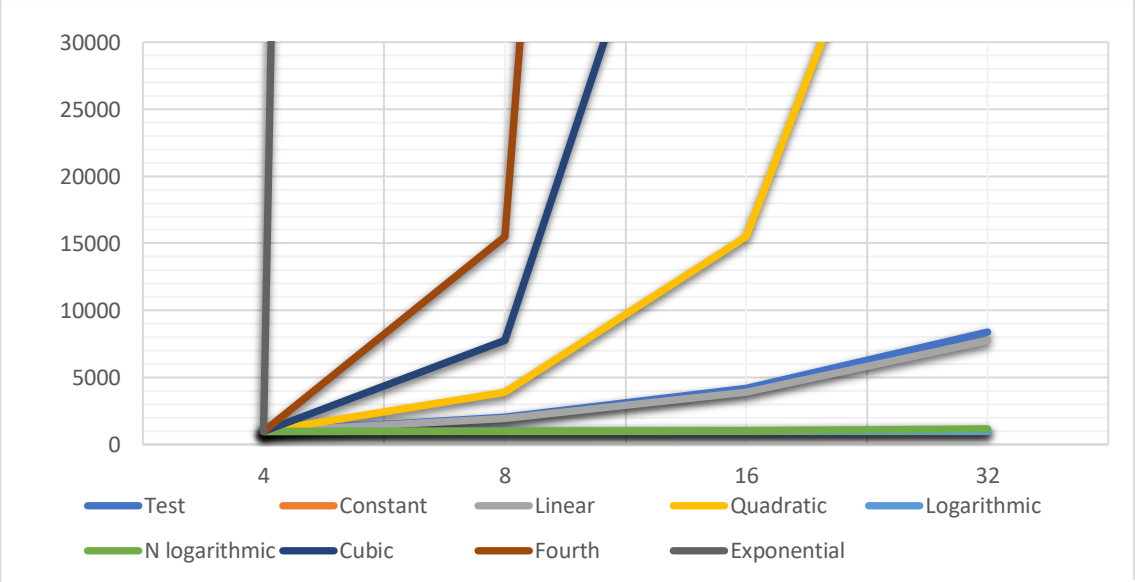

Figure 1 - comp value vs Complexity Order, weight = 150

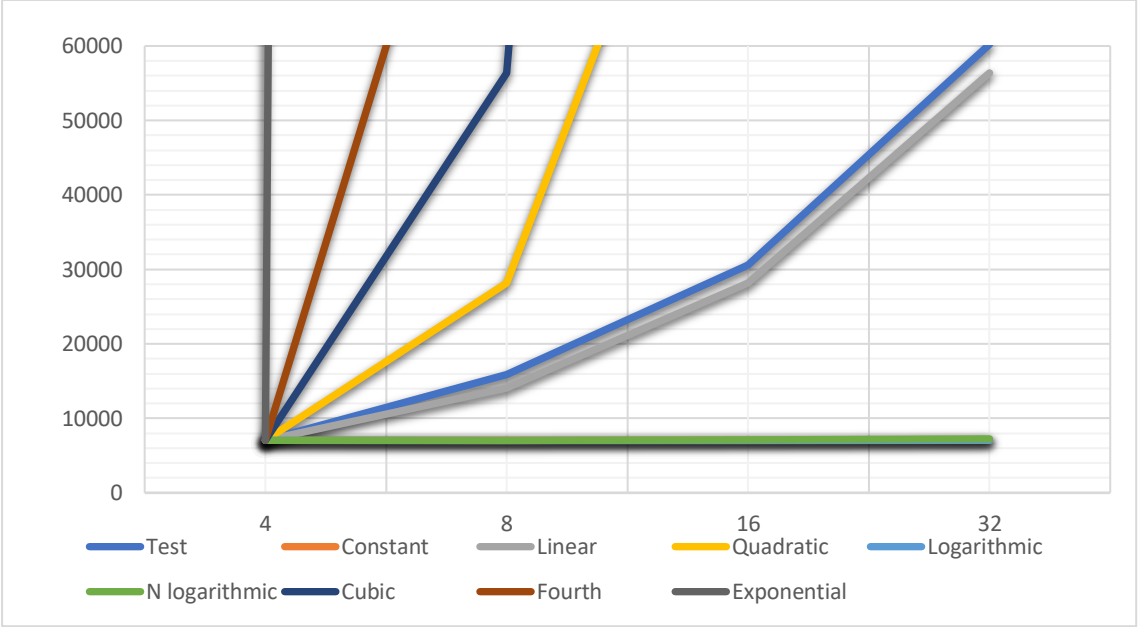

Figure 2 - comp value vs Complexity Order, weight = 1000

4. Discussion

During the algorithm complexity analysis, the biggest difficulty the group had was finding the correct instructions to count the number of comparisons done by the algorithm.

The knapsack capacity affects the number of comparisons the algorithm does, so we tried to find an expression to represent the number of comparisons. $2n \times W$ represents this attempt and by analyzing the graphic bellow (figure 3) we can see that the expression results are similar to the *comp* values obtained.

Input instance size (n)	***Comp Value*** ***Weight 1000***	***2n×W*** ***Expected Value***	***Comp Value*** ***Weight 150***	***2n×W*** ***Expected Value***
4	7948	8000	968	1200
8	15862	16000	2058	2400
16	30545	32000	4183	4800
32	60224	64000	8382	9600

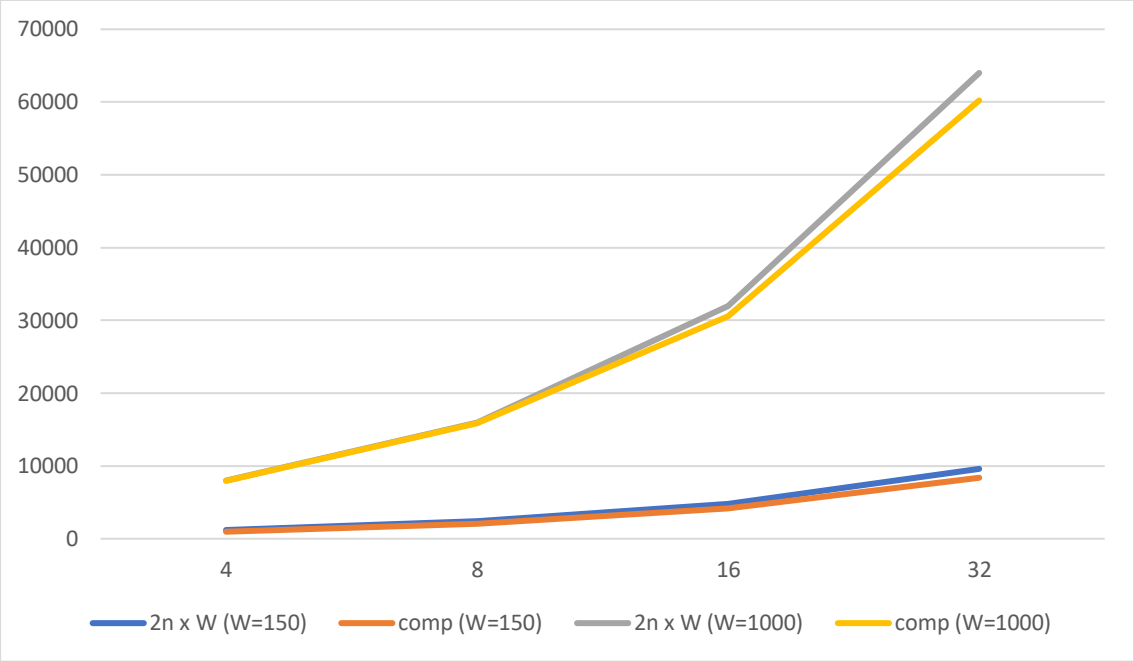

Figure 3 - Graph expression vs comp

These results indicate that the selected algorithm complexity order is linear (***O(n)***) because comp values and linear order values are most similar as showed in Table 1, Table 2, Figure 1 and Figure 2.

Doing the test batteries with only two weight values may be the biggest limitation of this analysis.

5. Conclusion and ongoing work

Since the beginning of the study of the Knapsack Problem many algorithms have surged and evolved.

In this report, we analyzed the complexity of an algorithm that would provide the optimal solution for the knapsack problem. We started by implementing the algorithm in the Java programming language, following the methodology of dynamic programming. After implementation two tests were done, each with 4 instances and the number of comparisons was recorded.

These number of comparisons obtained were compared with stand complexity functions values and, analyzing the graphs, we concluded that the chosen algorithm was of linear order.

Lastly, with the presented tests, we proposed a formula to calculate the approximated comparisons number that the algorithm will do to obtain the optimal solution.

Doing further test with different weight values may be the ideal approach to review this report.

With this report, we developed bibliographic research capabilities, algorithm implementation and performance analysis that will be useful in our student and working life.

Acknowledgments

The assistance provided by Professor Carlos Pereira was greatly appreciated and for the valuable data we used in our project.

References

1. Dantzig, T. (1930). *Numbers: The Language of Science.*
2. Knapsack Problem: Sove using Dynamic Programming. (n.d.). p. Example. Retrived from www.guru99.com/.
3- Mathews, G. B. (25 June 1897). "On the Partition of Numbers" (PDF). In M. G. B. Mathews, *Proceedings of the London Mathematical Society. 28* (pp. 486-490). doi: 10.1112/plms/s1-28.1.486.
4. *Operational Research, Algorithm Analysis and Optimization, supporting text, Escola Superior de Tecnologia e Gestão, 2021.* (n.d.).

Zhang, X. P. (2018). *J. Phys.: Conf. Ser. 1069 012024.*

Attachments

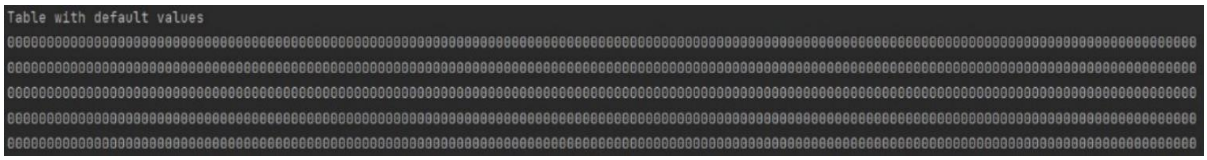

Figure 4 - Output Table

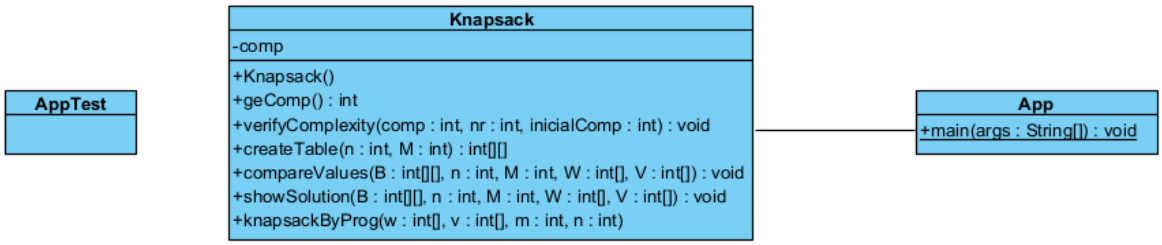

Figure 5 Class Diagram

Knapsack method:

```
public class Knapsack {
   private int comp;

   public Kanpsack() {
          this.comp = 0;
   }
   public int getComp() {
          return this.comp;
   }

   int[][] createTable(int n, int M) {
         int B[][] = new int[n + 1][M + 1];
         for (int i = 0; i <= n; i++) { // Create da table with defaut value 0
            for (int j = 0; j <= M; j++) {
               B[i][j] = 0;
            }
         }
         return B;
      }
      void compareValues(int B[][], int n, int M, int W[], int V[]) {
         for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= M; j++) {
               B[i][j] = B[i - 1][j];
               comp++; // Comp for first condicion
               if ((j >= W[i - 1])){
                  comp++; // Comp for second condicion
                  if ((B[i][j] < B[i - 1][j - W[i - 1]] + V[i - 1])) {
                     B[i][j] = B[i - 1][j - W[i - 1]] + V[i - 1];
                  }

               }

            }
         }
      }



   void showSolution(int B[][], int n, int M, int W[], int V[]) {

         System.out.println("Max Value:\t" + B[n][M]);

         System.out.println("Selected Packs: ");
```

```
        while (n != 0) {
          if (B[n][M] != B[n - 1][M]) {
            System.out.println("\tPackage " + n + " with W = " + W[n - 1] + " and Value = " + V[n - 1]);

            M = M - W[n - 1];
          }
          n--;
        }

        System.out.println("Comp: " + comp);

      }
    public void knapsackDyProg(int W[], int V[], int M, int n) {
        comp = 0;
        int B[][];
        B = creatTable(n, M);
        compareValues(B, n, M, W, V);
        showSolution(B, n, M, W, V);
        //verifyComplexity(comp,n,7926);
      }
```

MAIN:

```
public class App {
    public static void main(String[] args) {

    Knapsack algorithm = new Knapsack();
    final int M = 150;
    final int M2 = 1000;

    // Tests

    // n = 4
    int[] W = {18, 22, 10, 10};
    int[] V = {232,348,130,23};
    int n = V.length;

    int[] T2W = {18,22,10,10};
    int[] T2V = {232,348,130,23};
    int T2n = T2V.length;
```

```
// n = 8
int[] W2 = {22, 28, 120, 90, 10, 25, 6, 34};
int[] V2 = {18, 32, 110, 80, 7, 23, 8, 80};
int n2 = V2.length;

 int[] T2W2 = {18,22,10,10,18,42,20,14};
 int[] T2V2 = {232,348,130,230,180,113,111,111};
 int T2n2 = T2V2.length;

// n = 16
int[] W3 = {22, 28, 120, 90, 10, 25, 6, 34, 22, 26, 100 ,80, 34, 45, 56, 32};
int[] V3 = {18, 32, 110, 80, 7, 23, 8, 80, 19, 33, 105, 70, 12, 23, 28, 1};
int n3 = V3.length;

 int[] T2W3 = {18,22,50,40,18,160,20,14,232,38,130,230,180,113,111,111};
 int[] T2V3 = {832,348,130,230,480,413,231,111,232,348,10,30,480,213,211,561};
 int T2n3 = T2V3.length;

// n = 32
int[] W4 =
    {18,32,110,80,7,23,8,80,19,33,105,70,12,23,28,1,
    18,32,110,70,76,25,85,5,9,35,15,50,12,23,48,20};
int[] V4 =
    {22,28,120,90,10,25,6,34,22,26,100,80,34,45,56,32,
    12,23,130,30,20,15,6,14,62,76,130,80,44,25,26,92};
int n4 = V4.length;

 int[] T2W4 =
    {18,22,50,40,18,60,20,14,232,348,130,136,190,113,111,144,
    148,24,55,45,18,65,20,14,252,648,130,230,180,113,111,141};
int[] T2V4 =
    {332,348,130,230,480,413,231,111,232,348,10,30,480,213,211,561,
    842,348,123,330,480,313,281,411,332,248,101,20,380,213,211,221};
int T2n4 = T2V4.length;

// Tests with M = 150
algorithm.knapsackDyProg(W, V, M, n);
algorithm.knapsackDyProg(W2, V2, M, n2);
algorithm.knapsackDyProg(W3, V3, M, n3);
algorithm.knapsackDyProg(W4, V4, M, n4);

// Tests with M2 = 1000
algorithm.knapsackDyProg(T2W, T2V, M2, T2n);
algorithm.knapsackDyProg(T2W2, T2V2, M2, T2n2);
```

```
    algorithm.knapsackDyProg(T2W3, T2V3, M2, T2n3);
    algorithm.knapsackDyProg(T2W4, T2V4, M2, T2n4);
   }
}
```

Standard Complexity Functions[4]

Input size of the instances in increasing order (duplicating):

$$n \xrightarrow{\times 2} 2n \xrightarrow{\times 2} 4n \xrightarrow{\times 2} 8n \xrightarrow{\times 2} 16n \xrightarrow{\times 2} \dots$$

Proof: $O(1) \rightarrow 1$

$$comp \xrightarrow{\times 1} comp \xrightarrow{\times 1} comp \xrightarrow{\times 1} comp \xrightarrow{\times 1} comp \xrightarrow{\times 1} \dots$$

Proof: $O(n) \rightarrow 2n$

$$comp \xrightarrow{\times 2} 2comp \xrightarrow{\times 2} 4comp \xrightarrow{\times 2} 8comp \xrightarrow{\times 2} 16comp \xrightarrow{\times 2} \dots$$

Proof: $O(n^2) \rightarrow (2n)^2 = 2^2n^2 = 4n^2$

$$comp \xrightarrow{\times 4} 4comp \xrightarrow{\times 4} 16comp \xrightarrow{\times 4} 64comp \xrightarrow{\times 4} 256comp \xrightarrow{\times 4} \dots$$

Proof: $O(log_2 n) \rightarrow log_2(2n) = log_2 2 + log_2 n = 1 + log_2 n$

$$comp \xrightarrow{+1} 1 + comp \xrightarrow{+1} 2 + comp \xrightarrow{+1} 3 + comp \xrightarrow{+1} 4 + comp \xrightarrow{+1} \dots$$

Proof: $O(nlog_2 n)$
$\rightarrow (2n)log_2(2n) = (2n)(log_2 2 + log_2 n) = (2n)(1 + log_2 n) = 2n + 2nlog_2 n$

$$comp \xrightarrow{>(+2n)} 2n + comp \xrightarrow{>(+2n)} 4n + comp \xrightarrow{>(+2n)} 6n + comp \xrightarrow{>(+2n)} 8n + comp \xrightarrow{>(+2n)} \dots$$

Proof: $O(n^3) \rightarrow (2n)^3 = 2^3n^3 = 8n^3$

$$comp \xrightarrow{\times 8} 8comp \xrightarrow{\times 8} 64comp \xrightarrow{\times 8} 512comp \xrightarrow{\times 8} 4096comp \xrightarrow{\times 8} \dots$$

Proof: $O(n^4) \rightarrow (2n)^4 = 2^4n^4 = 16n^4$

$$comp \xrightarrow{\times 16} 16comp \xrightarrow{\times 16} 256comp \xrightarrow{\times 16} 4096comp \xrightarrow{\times 16} 65536comp \xrightarrow{\times 16} \dots$$

Proof: $O(2^n) \rightarrow 2^{(2n)} = 2^{(n2)} = (2^n)^2$

$$comp \xrightarrow{\wedge 2} comp^2 \xrightarrow{\wedge 2} comp^4 \xrightarrow{\wedge 2} comp^8 \xrightarrow{\wedge 2} comp^{16} \xrightarrow{\wedge 2} \dots$$

Variable Table

comp	the counting variable associated with the number of comparisons that occur at the runtime of the algorithm
n	number of items
W	an array of integers populated with the weight of each item
V	an array of integers populated with the weight of each item
M	maximum weight limit of the kanpsack